VMM Verification Methodology Manual 活用テクニック

赤星博輝

第4回　通知サービスとチャネルの使い方

これまでに，VMM（Verification Methodology Manual for SystemVerilog）では情報のやり取りを行うものとして，vmm_channelを紹介しました．このvmm_channelはデータのやり取りと同期を同時に行うことができる使いやすいものでした．今回は，vmm_channelの使い方をさらに深めるとともに，データ以外の情報をやり取りするvmm_notifyについて紹介します．（筆者）

VMM（Verification Methodology Manual for System Verilog）では，データを通信するために`vmm_channel`（チャネル）が用意されています．

● vmm_channelの復習

`vmm_channel`では，通信したいデータのクラスを`vmm_data`から継承して作成すれば，マクロ`` `vmm_channel``を使うことで簡単に定義できます．このマクロを使ってチャネルを定義すると，チャネルの名前は，

データ・クラス_channel

となります．**リスト1**に示す例は，データ・クラス`xy_dat`のチャネルをマクロを使って作成したので，チャネル名は`xy_dat_channel`となります．

このチャネルを使用するには，**リスト1**に示すように，`new`を用いてインスタンスを作成する必要があります．インスタンスを作成すると通信路として使用できます．データを送信する場合には`put`，データを受信する場合には`get`を使用してデータをやり取りします．

ここで重要な点は，`get`ではチャネルにデータが存在すればそのデータを受け取り次の実行に進みますが，チャネルにデータがない場合にはチャネルにデータが`put`されるのを待つことになるということです．データの送受信だけでなく，同期も同時に行っていることを頭に入れておきましょう．

● vmm_channelの使い方

VMMの特徴は，`vmm_channel`を使ってトランザクタを自由に接続できることです．これにより再利用が容易になり，最小の工数で最大のテスト・ベクタを作成できます．

リスト2のコードを実行すると，**図1**に示すような動作をします．時刻10に`put`を開始するのですが，実は時刻25に`put`は完了します．

この理由は，チャネルのバッファがいっぱいになってい

リスト1　チャネルの定義方法
`vmm_data`から作成したクラスであれば，`` `vmm_channel``というマクロを呼び出すだけで，通信するチャネルの定義が完了する．

```
class xy_dat extends vmm_data;
  static vmm_log log=new("XY_dat", "class");
  rand logic[7:0] mX,mY;
  function new( );
     super.new(log);
  endfunction

  // 必要なfunction,taskを定義する

endclass

`vmm_channel(xy_dat) ← クラスxy_datを通信するための
                        チャネルを定義
```

KeyWord　VMM，SystemVerilog，vmm_channel，チャネル，インスタンス，トランザクタ，ブロッキング，peek，sneak，notify

るとputしようとしてもできないので，バッファに空きがでるのを待つためです．**図2**のコードでは処理の速度が遅いため，データ送信側（タスクgen）の方が待たされることになります．

データ送信側（タスクgen）がputしたデータを，データ受信側（タスクcon）がgetして使用すると，そのgetした時点でデータ送信側のputが動作を再開します．これは，データ受信側がチャネルからデータを取り出すことでバッファに空きが生じるため，データ送信側のタスクの処理が再開されることになります．

vmm_channelの使い方をさらに深める

図2のように，データ受信側の処理が完了してから次のデータ送信処理を開始するには，どうしたらよいのでしょうか．このようなモデルをブロッキング完了モデルといいます．

● ブロッキングなどに利用できるpeek

ブロッキング完了モデルのような場合のために，チャネルにはpeekと呼ばれるtaskが用意されています．peekはチャネルからデータを取り出すことなく，データを見ることを可能とします（peekとは，のぞき見するという意味）．

リスト3に示すように，これまで受信側タスクでgetを1回呼んでいた処理を，peekとgetの2回に分けて呼び出すことにします．**図3**のように，最初にデータ受信側でpeekを呼び出すと，チャネルにデータを残したままデータを参照できるので，そのデータを用いた処理を行うことが

リスト2
チャネルをインスタンスし使用する例
チャネルを利用することで，タスクやトランザクタ間で通信を容易に行える．

```
xy_dat_channel      dchan=new("XY_channel","u0");   ← チャネルのインスタンス作成
task  gen;
   xy_dat  t;
    for (int i=0; i  < 10; i++ )  begin
        t= new;
        t.randomize( );
       #5  dchan.put(t);
    end
endtask                                              ← データ生成側タスク

task con;
 xy_dat  t;
    for (int i=0; i  < 10; i++ )  begin
     dchan.get(t);
       #20   d.display("sample data=" );
    end
endtask                                              ← データ受信側タスク

initial begin
     fork
           gen ();
           con();
     join
end
```

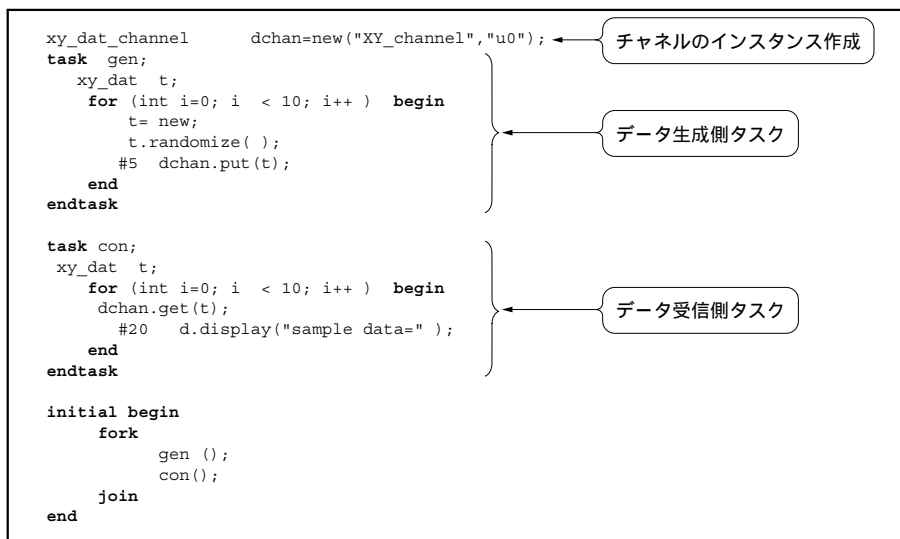
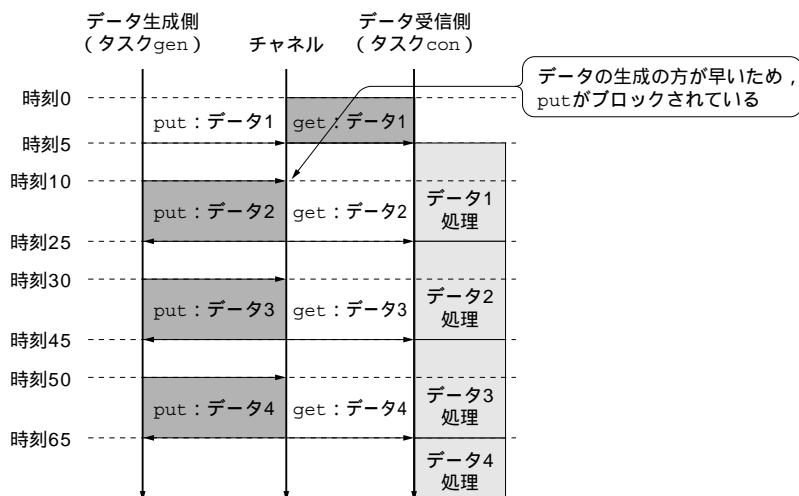



図1
チャネルを用いた通信のタイミング
getされないとチャネルにバッファが空かないため，データ生成側はputで待ちが発生する．

可能となります．この時点では，チャネルからデータが取り出されていないため（チャネルのバッファが1で定義されている場合），データ生成側の処理はブロッキングされて，putで停止しています．

受信側の処理が完了した段階でgetを使用すると，チャネルからデータを取り出します．データを取り出すことで，データ送信側のputのブロッキングが解除され，次のデータ送信処理が進められることになります．

HDLの記述では，ブロッキング代入とノンブロッキング代入の二つが重要なポイントとなりますが，ブロッキングとノンブロッキングがVMMでも利用できることが分かります．検証環境を構築するときに，必要に応じて選択することになるので，頭に入れておきましょう．

● **vmm_channelのバッファ・サイズの変更**

ブロッキングとノンブロッキングのほかに検証環境で重要なものとして，入力や出力のバッファ・サイズがあります．実はvmm_channelはデフォルトでバッファ・サイズが1になっています．先ほどのpeekを使う例では，バッファ・サイズが1のままpeekを使うことでブロッキングを実現していました．しかし，実際にはバッファ・サイズが1ではない検証環境が必要な場合もあります．

図2　ブロック完了モデル

データ受信側の処理が完了してから，次のデータを生成し，次のデータを生成するためにputでブロッキングしたい場合には，リスト2の方法ではできないことが分かる．

バッファのサイズを変更するためには，vmm_channelをインスタンス時（newを呼び出すとき）に第3引き数にサイズを指定するか，reconfigureの第1引き数にサイズを入れて再構成する必要があります．

例えば，バッファ・サイズを3にするためには，

```
xy_dat_channel  dat_chan
= new("name", "instance", 3);
```

のようにnewを使ってインスタンス作成時に設定するか，

```
dat_chan.reconfigure(3);
```

のようにreconfigureを用いて変更することができます．

リスト2のコードを，バッファ・サイズだけ変更して実行すると，**図4**の動作になります．vmm_channelではこのようにバッファのサイズを変更することができ，さまざま

図3　peekを用いた場合の動作

peekを使うことでチャネルからデータが取り除かれないため，データ生成側はputで停止したままになり，getでデータを取り除くと，次の処理を開始する．

リスト3
peekを用いたブロッキング

peekを使うと，チャネル中のデータをのぞき見(peek)することができる．これにより，チャネルにデータを残したまま処理すると，送信側の処理をブロッキングできる．

```
/*  peekを用いた処理に変更  */
task con;
 xy_dat  t;
    for (int i=0; i  < 10; i++ )  begin
            dchan.peek(t);
       #20   d.display("sample data=" );
          dchan.get(t);
    end
endask
```

peekではチャネルにデータを残したまま
データを読み出すことが可能

処理が終わった段階でgetを使用して
データをチャネルから取り除く

な検証の状態を作り出すことが可能です．

● こっそり書き込むsneak

peakが“こっそりのぞき見する”taskでしたが，“こっそり書き込む”sneakというfunctionもあります(sneakとは，「こっそり入る」という動詞)．チャネルのバッファが満杯であったとしても，チャネルにデータを入れることができます．

デザインを監視するモニタなどでは，基本的にデータをすべて記録する必要があります．このような場合には，sneakを使うことでバッファがフルかどうか考えることなく，処理を継続できます．

バッファを大きくするという手法も使えますが，バッファをいくら大きくしても安全ということはありえません(VMMは再利用可能な検証環境を構築することがポイントのひとつだが，別のプロジェクトで再利用するときにも安心なバッファの数などは分からない)．

また，ここで「なぜpeekはtaskで，sneakはfunctionなのか？」という疑問が出るかもしれません．

functionはゼロ遅延であり，taskは時間を持つ処理を記述するものです．よって，sneakはゼロ遅延で，確実にチャネルにデータを投入するためのものになります．

データ以外の情報を渡すnotify

ここまで，データのやり取りを中心に行うvmm_channelを説明してきましたが，これだけでは十分でない場合もあります．例えば，データやトランザクタの状態を知りたい場合などがありますが，その状態をvmm_channelですべてを通信しようとすると，テストベンチがくもの巣のようになって，再利用などできなくなってしまいます．

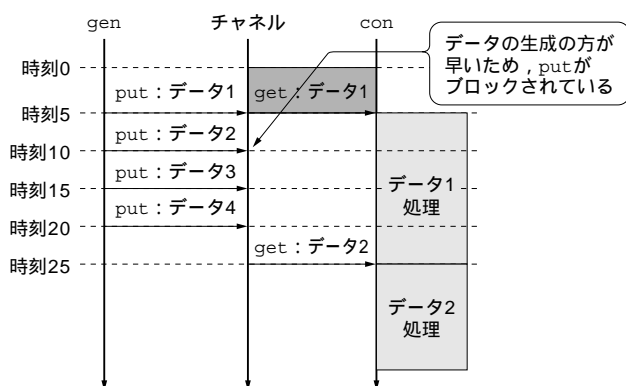



図4　チャネルのバッファを3個にした場合の動作
バッファを3個にすることで，バッファにデータがフルになるまでputすることができる．

VMMでは，データ以外の情報を渡す方法としてvmm_notifyが用意されています．このvmm_notifyには，以下の三つの同期モードがあります．

- ONE_SHOT：通知の指示を待っているスレッドのみが通知を受ける
- BLAST：通知が指示されるときの同じステップで，通知の指示を待っているすべてのスレッドが通知を受ける．
- ON_OFF：ON，OFFのレベルで通知を行い，明示的にリセットするまで通知が持続される．

ここでは，ONE_SHOTとON_OFFの使用方法を説明します．

● ONE_SHOTは1時点で有効なイベントを発生

ONE_SHOTのnotifyでは，ある1時点で有効なイベントを発生することができます．そのイベントを受け取るのは，そのイベントが発生する前にイベント待ちに入ったものになります．その通知イベントを使うためには，以下のステップが必要になります．

1) configureによって新しい通知を定義する．
2) wait_forでイベント待ちを行う．
3) indicateでイベントを発生させる．

簡単なサンプルを**リスト4**に示します．

vmm_notifyを使って新しい通知を定義する場合には，まずメッセージ・サービスを定義しておきます．これは，vmm_notifyでもメッセージ・サービスを利用するためです．次に通知サービスであるvmm_notifyのnt1を作成します．実際にイベントを使うためにはこのnt1に対してconfigureをして，新しい通知(イベント)を作成します．ここで作成された通知(イベント)は識別子(ID)で管理されるので，configureの返り値を覚えておく必要があります．

この変数t1をIDに持つイベントを待つには，

```
通知サービス.wait_for(識別子)
```

と記述します．この場合は通知サービスがnt1，識別子がt1で管理されているので，

```
nt1.wait_for(t1)
```

となります．

この変数t1をIDに持つイベントを発生させるためには，

```
通知サービス.indicate(識別子)
```

とします．イベント待ちと同様に通知サービスがnt1，識別子がt1で管理されているので，

```
nt1.indicate(t1)
```

で，イベントを発生させることができます．

このサンプルの実行結果は，

```
CHECK1: ONE_SHOT            100
```

になります．

図5に示すように，CHECK1ではイベントを発生する前にwait_forで待ちに入ると，その後のindicateで待ちが解除されます．また，CHECK2では，wait_forを呼んだ後にまだindicateが呼ばれていないため，次のイベントを待っている状態になります．

● ON_OFFは継続するON/OFFの状態を使った通知

ONE_SHOTはイベントを受け渡しするために利用しますが，状態を扱うには不適当です．そのため，状態を持つON_OFFを用いたvmm_notifyを紹介します．

ON_OFFの通知イベントを使うには，以下のステップが必要になります．

1）configureによって新しい通知を定義する．
2）wait_forでイベント待ちを行う．
3）indicateでイベントを発生させる．

簡単なサンプルを**リスト5**に示します．

ONE_SHOTと同様に通知サービスnt1を作成します．このとき，メッセージ・サービスを登録するところもONE_SHOTと同じです．通知サービスnt1に対してconfigureで，ON_OFFの新しい通知を作成します．このとき，ON_OFFの通知も識別子(ID)を用いて区別されるので，変数にその識別子を保存しておく必要があります．

このON_OFFの通知は，ONを待つ場合には，

```
通知サービス.wait_for(識別子)
```

を使用しますし，OFFを待つ場合には，

```
通知サービス.wait_for_off(識別子)
```

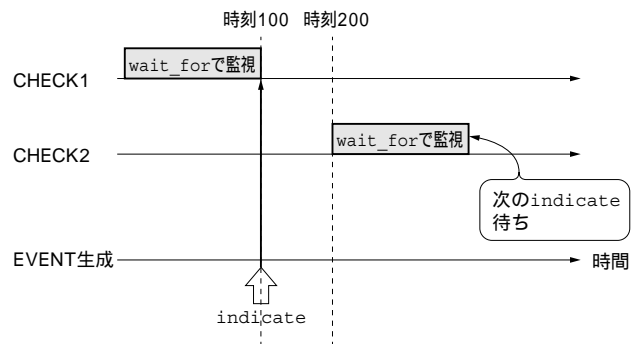



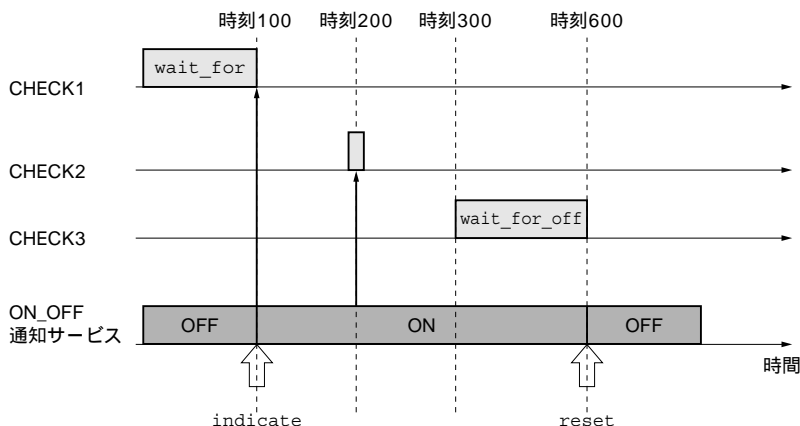
図5　ONE_SHOTの動作イメージ
wait_forでONE_SHOTイベントを待っていると，indicateによるイベントにより待ちを解除する．

リスト4　ONE_SHOTのvmm_notifyの使用例
wait_forを使うことでイベントを待ち，indicateを用いてイベントを発生させられる．

```
vmm_log log=new("NOTIFY", "class");

vmm_notify nt1=new(log);

int t1;

initial begin
  t1 = nt1.configure (, vmm_notify::ONE_SHOT);
 #100      nt1.indicate(t1);
 #1000     $finish;
end
initial begin   /* CHECK1 */
        nt1.wait_for(t1);
        $display("CHECK1: ONE_SHOT %t",$time);
end

initial begin   /* CHECK2 */
  #200  nt1.wait_for(t1);
        $display("CHECK2: ONE_SHOT %t",$time);
end
```

リスト5
ON_OFFのvmm_notifyの使用例
ONを待つにはwait_for**，OFFを待つには**wait_for_off**を利用する．ONにするには**indicate**，OFFにするには**reset**を使用する．**

```
vmm_log log=new("NOTIFY", "class");
vmm_notify nt1=new(log);
int t1;
initial begin
  t1 = nt1.configure (, vmm_notify::ON_OFF);
 #100  nt1.indicate(t1);
 #500  nt1.reset(t1);
 #4500 $finish;
end

initial begin /*CHECK1*/
   nt1.wait_for(t1);
   $display("CHECK1: ON %t",$time);
end
initial begin  /*CHECK2*/
  #200   nt1.wait_for(t1);
         $display("CHECK2: ON %t",$time);
end

initial begin  /*CHECK3*/
  #300   nt1.wait_for_off(t1);
         $display("CHECK3: OFF %t",$time);
end
```


図6
ON_OFFの動作イメージ
ON_OFFは状態なので，ON(OFF)状態であれば，wait_for(wait_for_off)**は即座に待ちが解除される．**

を使用します．また，すでにON(OFF)の場合にwait_for(wait_for_off)を呼ぶと，すぐに待ちが解除されます．

リスト5を実行したイメージが**図6**です．CHECK1はONになる前からwait_forを開始し，ONになったときに待ちが解除されています．CHECK2はONになってからwait_forを呼び出していて，即座に待ちが解除されています．CHECK3はOFFを待って，resetが呼ばれるとOFFになるので，待ちが解除されることになります．

● チャネルにはnotifyイベントがセットされる

VMMのライブラリでは，notifyを用いた通知がいろいろなところに埋め込まれています．

例えば，vmm_channelでは，FULL，EMPTY，PUT，GOT，PEEKED，ACTIVATED，ACT_STARTED，ACT_COMPLETED，ACT_REMOVED，LOCKED，UNCLOCKEDといったイベントがあらかじめ定義されています．ユーザはこれらを使用してチャネルの情報を取得することができます．

このイベントを拾うプログラムを**リスト6**に示します．これにより，チャネル上のイベントを利用して処理を行うことも可能になります．

チャネルと通知の連携

vmm_channelには，あらかじめメソッドに通知が埋め込まれています．ここで，vmm_channelのメソッドをさらに四つ紹介します(**リスト7**)．

vmm_channelには，アクティブ・スロットという考え方があります．vmm_channelで有効になっているデータと考えればよいと思います．

- activte：先頭のデータをアクティブ・スロットに入れ，アクティブ・スロットの状況をPENDINGにします．

リスト6　チャネルにはvmm_notifyが組み込まれている

vmm_channelは動作ごとに通知が発生されるようにあらかじめ作成されており，この例ではpeek，get，putのイベントを受け取ることが可能．

```
initial begin
  while(1) begin
    dchan.notify.wait_for(vmm_channel::PEEKED);  ← チャネルがpeekされるのを待つ
    $display("NOTIFY:PEEKED @%t", $time);
  end
end

initial begin
  while(1) begin
    dchan.notify.wait_for(vmm_channel::GOT);  ← チャネルがgetされるのを待つ
    $display("NOTIFY: GOT @%t", $time);
  end
end

initial begin
  while(1) begin
    dchan.notify.wait_for(vmm_channel::PUT);  ← チャネルがputされるのを待つ
    $display("NOTIFY: PUT @%t", $time);
  end
end
```

リスト7　チャネルの操作を詳細化

vmm_channelではチャネルの状況の詳細を示すメソッドを使用することで，より詳しい状態を通知できる．

```
for(int  i=0;i<10;i++) begin
        dchan.activate(mydat);
   #20  dchan.start();
   #60  dchan.complete();
   #20  dchan.remove();
end
```

リスト8
詳細の通知を受け取る記述例

vmm_channelの状況を外部から検出することが可能であり，この通知を利用し，あとから，さまざまな処理を組み込むことができる．

```
initial begin
  while(1) begin
    dchan.notify.wait_for(vmm_channel::ACTIVATED);  ← acitvate( )を待つ
    $display("NOTIFY: ACTIVATED @%t", $time);
  end
end
initial begin
  while(1) begin
    dchan.notify.wait_for(vmm_channel::ACT_STARTED);  ← start( )を待つ
    $display("NOTIFY: ACT_STARTED @%t", $time);
  end
end
initial begin
  while(1) begin
    dchan.notify.wait_for(vmm_channel::ACT_COMPLETED);  ← complete( )を待つ
    $display("NOTIFY: ACT_COMPLETED @%t", $time);
  end
end
initial begin
  while(1) begin
    dchan.notify.wait_for(vmm_channel::ACT_REMOVED);  ← remove( )を待つ
    $display("NOTIFY: ACT_REMOVED @%t", $time);
  end
end
```

- start：アクティブ・スロットの状況をSTARTEDにします．
- complete：アクティブ・スロットの状況をCOMPLETEDにします．
- remove：アクティブ・スロットの状況をINACTIVEにし，データをチャネルから取り除きます．

この四つを使って，**リスト3**のブロッキング処理を書き換えたのが**リスト8**です．こうすることで，イベントが発生したときの処理を記述したり，ログを出力したりすることが容易にできます．

こういった点は，通常の検証環境を作成する場合には作り込みが難しいところですが，VMMではこういった機能が埋め込まれており，少しずつテスト・ベンチを高度化することができます．

あかぼし　ひろき
(株)ソリューション・デザイン・ラボラトリ

<筆者プロフィール>
赤星博輝．ハードウェアの検証とソフトウェアのテストの融合が現在のテーマです．ハードウェアではVerification Methodology ManualとSystemVerilogを推進し，ソフトウェアではRTOSを中心に活動中です．